**Politechnika Świętokrzyska**

**Wydział Elektrotechniki, Automatyki i Informatyki**

**Szymon Kołodziejczyk**

**Numer albumu: 091277**

**Tytuł pracy dyplomowej w języku polskim**

**Praca dyplomowa   
na studiach I-go stopnia**

**na kierunku Informatyka**

Promotor pracy dyplomowej:

dr inż. Grzegorz Łukawski

Wydział Elektrotechniki, Automatyki i Informatyki

Konsultant pracy dyplomowej:

##### Tytuł naukowy, imię i nazwisko konsultanta

#### **Kielce 2023**

#### **Tutaj ma być zamieszczona kopia (skan) zadania na pracę**



#### 

Strona 7

Oświadczenie autora pracy – jeśli jest wymagane przez promotora



**Silnik dla gier 2D z otwartym światem**

**Streszczenie**

Streszczenie pracy – 4 do 10 wierszy

Celem pracy jest projekt i implementacja silnika dla gier 2D z otwartym światem, z możliwością modyfikacji otoczenia (niszczenia, budowania) oraz z elementami fabularnymi (np. zadania, walka z wrogami, handel z postaciami niezależnymi). Proceduralne (pseudolosowe) generowanie elementów gry.

Słowa kluczowe: - do 6

**2D open world game engine**

**Summary**

Summary of the thesis in English - 4 to 10 lines

The goal of the project is to design and implement a 2D game engine for an open-world environment, allowing for environment modification (destruction, construction), incorporating narrative elements (quests, combat with enemies, trading with non-playable characters). It will feature procedural (pseudo-random) generation of in-game elements

Keywords: - up to 6

Strona 8 lub 10 (tył streszczeń) – pusta, bez numeru - ten tekst należy usunąć

##### **SPIS TREŚCI**

Spis treści

[1. WSTĘP 13](#_Toc155364998)

[1.1 Gry z otwartym światem 13](#_Toc155364999)

[1.1 Opis elementów 14](#_Toc155365000)

[1.1.1 Wielkość mapy 14](#_Toc155365001)

[1.1.2 Ułatwienie przemieszczania się po świecie 14](#_Toc155365002)

[1.1.3 Nieliniowe podejście w zwiedzaniu 15](#_Toc155365003)

[1.2 Cel Pracy 15](#_Toc155365004)

[1.3 Zakres Pracy 15](#_Toc155365005)

[2. Projekt 16](#_Toc155365006)

[2.1 Wymagania funkcjonalne 16](#_Toc155365007)

[2.2 Wymagania niefunkcjonalne 16](#_Toc155365008)

[2.3 Mechaniki zaprojektowane w innych aplikacjach 17](#_Toc155365009)

[2.3.1 Przechwytywanie obiektów na planszy 17](#_Toc155365010)

[2.3.2 Przechowywanie danych odnośnie mapy i obiektów w plikach 18](#_Toc155365011)

[2.3.3 Poruszanie przeciwników 19](#_Toc155365012)

[3. Implementacja 20](#_Toc155365013)

[3.1 Wybór technologii 20](#_Toc155365014)

[3.1.1 Wybór języka 20](#_Toc155365015)

[3.1.2 Wybór biblioteki graficznej 21](#_Toc155365016)

[3.1.3 Wybór środowiska programowania 21](#_Toc155365017)

[3.2 Implementacja Drzewa Czwórkowego. 22](#_Toc155365018)

[3.3 Implementacja Systemu zapisu 26](#_Toc155365019)

[3.4 Implementacja Systemu szukania drogi 27](#_Toc155365020)

[3.5 Generowanie terenu 29](#_Toc155365021)

[Bibliografia 34](#_Toc155365022)

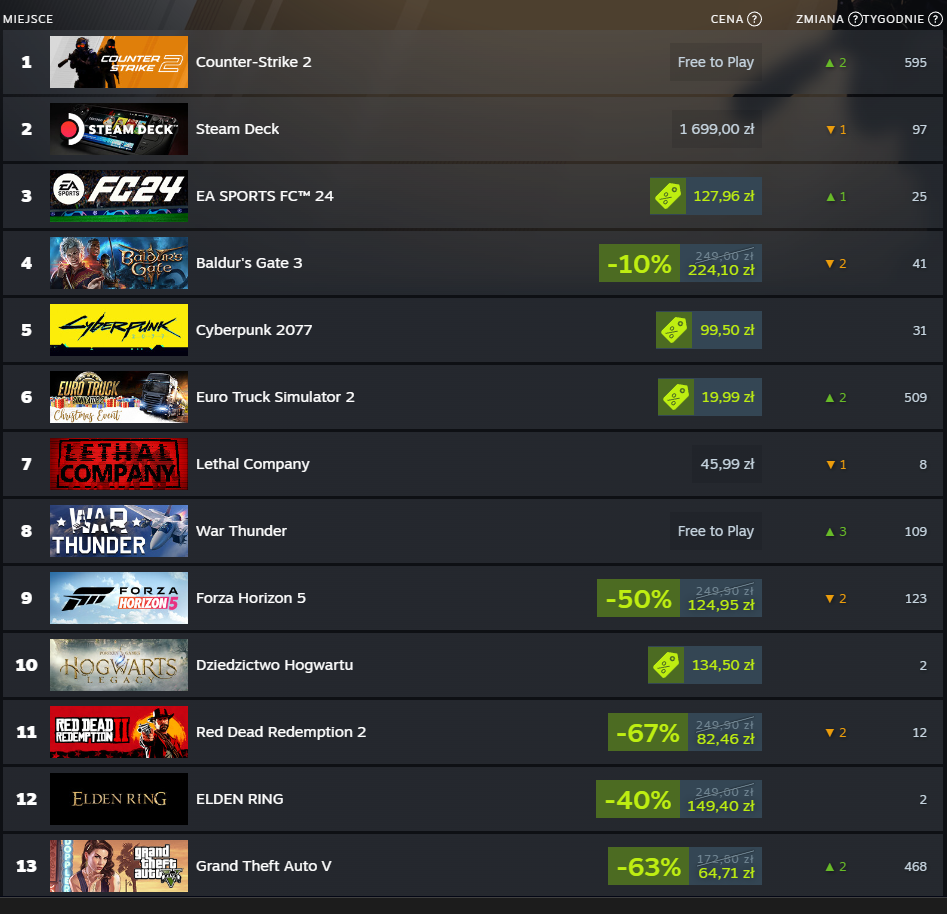
Strona 10 lub 12 - pusta lub kontynuacja spisu treści

1. WSTĘP

Rozdział 1 wprowadza czytelnika w świat gier z otwartym światem, prezentując charakterystyczne elementy tego gatunku oraz przykładowe tytuły.

* 1. Gry z otwartym światem

Gry z otwartym światem cechują się udostępnieniem graczowi dużej przestrzeni, w której może swobodnie się poruszać. Ostatnio ten gatunek gier staje się coraz popularniejszy i częściej wykorzystywany przez twórców. W notowaniach platformy Steam z okresu od 26 grudnia 2023 roku do 2 stycznia 2024 roku, wśród 12 najbardziej dochodowych produkcji, aż 6 gier zalicza się do kategorii gier z otwartym światem.



Rysunek 1.1.1 Bestsellery w tygodniu 100 najlepszych produktów według przychodu Od wt., 26 grudnia 2023 do wt., 2 stycznia 2024

* 1. Opis elementów

Gry oparte na otwartym świcie różnią się znacząco od siebie w stylizowaniu świata możemy się wcielić w średniowieczną postać, przebyć dziki zachód lub zwiedzać świat w samochodzie. Mimo tych różnic gatunek ten posiada kilka wspólnych elementów.

* + 1. Wielkość mapy

Gry z otwartym światem oferują graczom duże przestrzenie do eksploracji, umożliwiając im zwiedzanie rozległych i zróżnicowanych terenów. Istnieją dwa główne podejścia do tworzenia świata w tego typu grach.

Pierwszym z nich jest generowanie świata dynamicznie podczas rozgrywki gracza. Światy generowane w ten sposób powstają w uporządkowany, losowy sposób. Gry o takich światach posiadają olbrzymie mapy, które są tworzone na podstawie określonych wartości - tak zwanych "nasionek". Przykładami popularnych gier, które generują świat w ten sposób, są: Minecraft, Necesse, Terraria.

Drugim podejściem, bardziej pracochłonnym, jest stworzenie świata przez ludzi, poprzez ręczne umieszczenie wszystkich elementów na mapie. Świat taki nie jest generowany losowo - przy każdym rozpoczęciu gry mapa pozostaje taka sama. Przykłady gier, które używają tego podejścia, to: GTA 5, Elden Ring, Cyberpunk 2077, The Legend of Zelda: Tears of The Kingdom.

* + 1. Ułatwienie przemieszczania się po świecie

Gry te posiadają rozległe mapy, dlatego twórcy wprowadzili różne udogodnienia dla graczy, aby ułatwić im poruszanie się po świecie gry. Dodane rozwiązania mają na celu ułatwienie eksploracji, takie jak szybka podróż oraz dostęp do wierzchowców i pojazdów.

Szybka podróż jest zaprojektowana w taki sposób, aby gracze mogli łatwo powrócić do miejsc odwiedzonych wcześniej. W tym celu tworzone są obiekty umożliwiające teleportację, przyjmujące różne formy - od miejsc łaski w grze Elden Ring, po terminale w grze Cyberpunk 2077.

Wierzchowce, zazwyczaj konie, są powszechnie dostępne dla graczy, umożliwiając znacznie szybsze pokonywanie dużych odległości na mapie. Dodatkowo, w niektórych grach istnieją pojazdy, takie jak samochody czy motory, które można nabyć w świecie gry. Czasami deweloperzy gier decydują się także na wprowadzenie możliwości budowy własnych pojazdów. Przykładem takiego podejścia jest gra "The Legend of Zelda: Tears of the Kingdom".

* + 1. Nieliniowe podejście w zwiedzaniu

W grach z dużą otwartą mapą, twórcy zazwyczaj pozostawiają graczom swobodę eksploracji świata, nie nakazując precyzyjnie, dokąd muszą się udać, aby przeżyć przygodę. Zamiast tego, subtelnie wskazują pewne miejsca, które mogą być warte odwiedzenia. Jako gracze, mamy możliwość skorzystania z tych wskazówek, ale również cieszymy się dużą swobodą, ponieważ nie ma sztywnych blokad ograniczających nasze poruszanie się.

Dlatego też, istnieje możliwość eksploracji świata w sposób, który może różnić się od pierwotnych intencji twórców, umożliwiając graczom wyjątkowe i nieprzewidziane przygody. Brak wyraźnych ograniczeń pozwala na unikalne doświadczenia w grze, gdzie eksploracja może prowadzić do odkrywania nieoczekiwanych miejsc i wydarzeń.

* 1. Cel Pracy

Celem pracy jest zaprojektowanie silnika umożliwiającego tworzenie gier, które pozwolą na generowanie świata gry oraz rozbudowę tego świata poprzez dodanie pewnych elementów charakterystycznych dla gier typu "sandbox". Chcemy uwzględnić możliwość wpływania na teren, takie jak niszczenie bloków oraz budowanie, jak również implementację mechaniki walki z postaciami niezależnymi od gracza.

Silnik będzie oparty na zaawansowanych algorytmach generowania terenu, pozwalając graczom na eksplorację świata, który będzie unikalny przy każdym uruchomieniu gry. Dodatkowo, planujemy implementować system interakcji z otoczeniem, który umożliwi niszczenie oraz konstruowanie elementów otaczającego świata. Mechanika walki z postaciami niezależnymi będzie stanowić istotny element rozgrywki, zapewniając graczom wyzwania i interakcję z otoczeniem.

Silnik będzie dążył do zapewnienia elastyczności i możliwości rozszerzania gry przez dodanie nowych funkcji czy elementów. Ma umożliwić tworzenie gier, które dadzą graczom pełną swobodę eksploracji, tworzenia oraz walki w otwartym i interaktywnym świecie.

* 1. Zakres Pracy

Zakres pracy obejmuje stworzenie gry na platformę Windows, która umożliwi graczom eksplorację różnorodnych terenów generowanych losowo w oparciu o "nasienie" wprowadzone przez gracza podczas tworzenia postaci.

Gra będzie zapewniać unikalne doświadczenia każdemu graczowi, pozwalając na generowanie losowych światów na podstawie określonego "nasienia" wprowadzonego podczas tworzenia postaci. Gracz będzie mógł wprowadzić swoje własne parametry, a na ich podstawie zostanie wygenerowany świat gry, który będzie różnił się od innych świata wygenerowanych przez inne "nasienia".

Celem gry będzie umożliwienie eksploracji zróżnicowanych terenów w wygenerowanym losowo świecie, zapewniając unikalne doświadczenia zależne od wybranego "nasienia". Gracz będzie miał możliwość odkrywania i badania różnorodnych miejsc, stworzonych na podstawie parametrów wprowadzonych na początku gry.

2. Projekt

Rozdział 2 będzie opisywał zbiór kluczowych funkcji i mechanik, które mają być zawarte w silniku gry, a następnie wykorzystane do stworzenia rozgrywki w grze z otwartym światem.

2.1 Wymagania funkcjonalne

Rozgrywka:

* Generowanie losowych map na podstawie początkowej wartości gracza.
* Interaktywna destrukcja obiektów na mapie.
* Edycja i dodawanie nowych obiektów do mapy.
* Walka z różnorodnymi przeciwnikami.
* Handel z postaciami niezależnymi.
* Tworzenie różnorodnych przedmiotów w grze.
* Używanie i aplikowanie różnego rodzaju przedmiotów.
* Funkcja zapisu i odczytu stanu gry.

Elementy poza rozgrywką:

* Możliwość dodawania nowych przedmiotów do gry z poziomu menu gry.
* Dodawanie nowych schematów receptur do gry z poziomu menu gry.

2.2 Wymagania niefunkcjonalne

* Gra dostępna pod systemy windows 10, 11
* Odpowiednia optymalizacja by systemy posiadał niskie straty w klatkach na sekundę
* System powinien być prosty w dodawaniu nowych obiektów do rozgrywki
* Grafika powinna posiadać swój charakterystyczny element dla produkcji

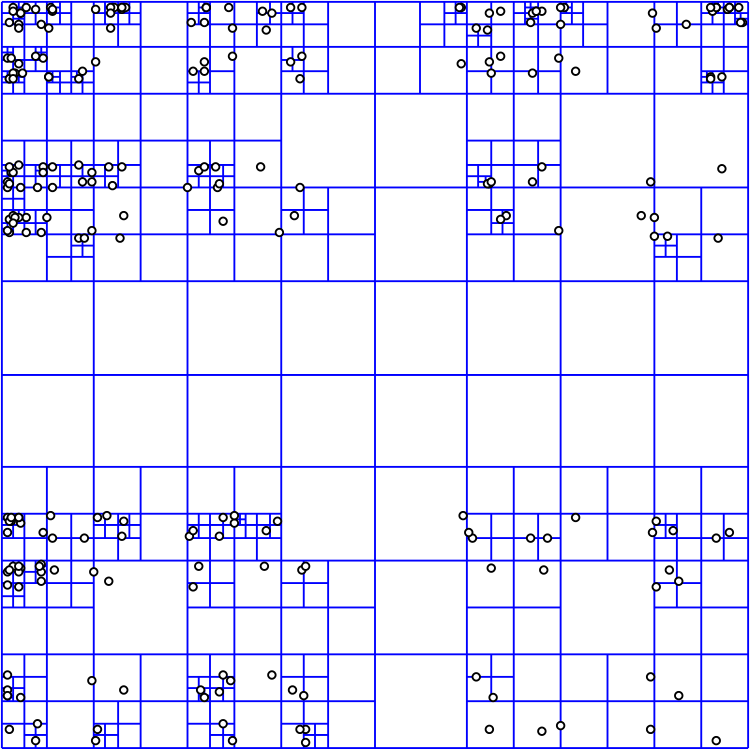
2.3 Mechaniki zaprojektowane w innych aplikacjach

W celu stworzenia pewnych mechanik w grze, korzystne jest przeanalizowanie funkcjonowania tych elementów w innych tytułach.

2.3.1 Przechwytywanie obiektów na planszy

Zbieranie obiektów na mapie może wydawać się proste, np. przechowywanie ich w liście lub wektorze. Jednak w przypadku dużej liczby obiektów, operacja odnalezienia odpowiedniego może stać się kosztowna. Przy podziale mapy na fragmenty o rozmiarze np. 1600x900 pikseli, gdy obszar ten zawiera 5 obiektów, szukanie w pierwszej ćwiartce planszy nie jest zbyt kosztowne - wymaga sprawdzenia jedynie 5 obiektów w tym obszarze. Problem pojawia się, gdy liczba obiektów na tym obszarze wzrośnie z 5 do 50. W takim przypadku odnalezienie wszystkich obiektów w tej części planszy wymaga 50 porównań pozycji obiektów. W przypadku, gdy trzy obiekty próbują uzyskać dostęp do tych danych w jednej pętli, liczba porównań pozycji obiektów wzrasta do 150.

Istnieje jednak rozwiązanie tego problemu - można skorzystać z struktury danych drzewa czwórkowego. Ta struktura działa następująco: przechowuje wszystkie obiekty i w przypadku, gdy liczba obiektów osiągnie wartość maksymalną(podaną w kodzie programu), dzieli obszary na mniejsze regiony i grupuje obiekty w tych obszarach.



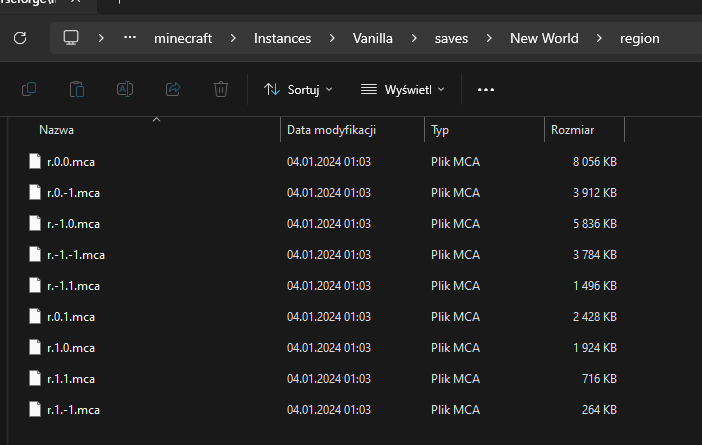
Rysunek 2.3.1.1 Drzewo czwórkowe wizualizacja struktury

Dzięki pogrupowaniu wszystkich obiektów na obszary z rysunku, możemy zauważyć, że aby uzyskać elementy z pierwszej ćwiartki, wystarczy dostarczyć do struktury drzewa obszar, z którego interesują nas obiekty. Następnie drzewo czwórkowe zwróci nam listę wszystkich poszukiwanych obiektów. Ta struktura działa na zasadzie sprawdzania obszarów: jeśli dany obszar nie odpowiada interesującemu nas obszarowi, pomija tę część drzewa. Proces wyszukiwania obiektów jest realizowany za pomocą funkcji rekurencyjnej.

2.3.2 Przechowywanie danych odnośnie mapy i obiektów w plikach

Przy zapisie wszystkich obiektów w jednym pliku pojawia się problem, który narasta wraz z dodawaniem nowych fragmentów mapy. Na początku, gdy plik zawiera tylko kilka fragmentów mapy, jego rozmiar nie jest znaczący. Kłopoty zaczynają się jednak, gdy generujemy setki kawałków terenu i próbujemy je zapisać w jednym pliku. Każde kolejne zapisywanie i wczytywanie terenu wymaga coraz więcej czasu, co drastycznie obniża wydajność i powoduje dyskomfort użytkownikowi.

Istnieje kilka potencjalnych rozwiązań tego problemu. Jednym z nich może być podział na mniejsze fragmenty, co pozwoliłoby zwiększyć wydajność, zwłaszcza przy obszarach o dużych rozmiarach. Podobny system plików został zastosowany w grze Minecraft, gdzie każdy plik zawierał kilkanaście fragmentów mapy.



Rysunek 2.3.2.1 Rysunek przedstawia katalog z zapisanymi regionami odpowiednio ponumerowanymi.

W losowo generowanych prawie nieograniczonych mapach gry podejście do przechowywania danych jest ważne i zaniedbanie tego elementu może prowadzić do sporych problemów w przyszłości.

**2.3.3 Poruszanie przeciwników**

Tworzenie logicznych zachowani dla ruchu przeciwników może być wyzwaniem. Załóżmy, że chcemy stworzyć przeciwnika, który będzie podążał za graczem, utrzymując dystans od niego na poziomie 400 pikseli. Rozwiązanie wydaje się proste w implementacji, jednak pojawia się problem w sytuacji, gdy na linii prostej pomiędzy przeciwnikiem a graczem znajduje się przeszkoda, przez którą nie można przejść. W takim przypadku przeciwnik będzie próbował dostać się przez tę przeszkodę, dopóki gracz nie zmieni swojej pozycji. Takie zachowanie sprawia, że gra staje się zbyt łatwa, ponieważ przeciwnicy nie są w stanie dotrzeć do gracza, gdy ten znajduje się za odpowiednią przeszkodą.

Istnieje kilka algorytmów poszukujących ścieżek od punktu A do punktu B, z których jednym z popularnych jest algorytm A\*. Ten algorytm analizuje kolejne pola, sprawdzając ile kroków potrzebuje, aby dotrzeć do celu, oraz ile czasu zajęło mu dotarcie do danej pozycji.



Rysunek 2.3.3 Algorytm A\* szukający ścieżki z białego kwadratu do żółtego punktu końcowego

Korzystanie z tego algorytmu pozwala przeciwnikom na bardziej inteligentne poruszanie się po mapie, co utrudnia ich zablokowanie. To stanowi większe wyzwanie dla graczy w aspekcie ucieczki oraz starć z nimi.

3. Implementacja

Rozdział 3 opowiada o procesie implementowania rozwiązań oraz wybór technologii w których postanie silnik.

3.1 Wybór technologii

Do stworzenia silnika do gier potrzebne jest wybranie środowiska w którym powstanie. Pośród języków programowania istnieje duża gamma możliwości.

3.1.1 Wybór języka

Ponieważ silniki do gier muszą zapewniać wysoką wydajność, języki interpretowane są wykluczone z listy preferowanych narzędzi do tworzenia rozbudowanych gier i silników gier. Zamiast tego preferowane są języki kompilowane, których kod jest przekształcany na język bardziej zbliżony do instrukcji komputerowych. Na rynku popularnymi językami są Java, C++, C#, C oraz Rust.

Porównując dostępne opcje, Java jest atrakcyjna, jeśli celem jest wydanie produktu na różne platformy. Jednakże ma pewne wady, takie jak automatyczne zwalnianie pamięci RAM w momencie, gdy obiekt nie jest dostępny w kodzie. To rozwiązanie, choć skuteczne, może prowadzić do niewłaściwego zarządzania pamięcią RAM, szczególnie gdy proste programy zajmują znacznie więcej pamięci, niż powinny, po jej zapełnieniu.

C, najstarszy język w zestawieniu, ma swoje ograniczenia wynikające z długiej historii. Brak obiektowości stwarza pewne problemy, wymaga poszukiwania alternatywnych rozwiązań. Jednak jego głównym atutem jest niski stopień abstrakcji, co czyni go najszybszym spośród wymienionych języków.

Rust to nowa, ciekawa opcja na rynku, mająca zaledwie 8 lat. Posiada kilka przydatnych mechanizmów, ale nauka tego języka może być wyzwaniem ze względu na odmienny charakter w porównaniu do tradycyjnych języków obiektowych.

Z dwóch języków, które są sukcesywnie wykorzystywane w grach i silnikach do gier, C++ jest szeroko stosowany w silniku Unreal Engine, podczas gdy C# jest wykorzystywany w silniku Unity. Analizując języki, w jakich napisane zostały te dwa popularne silniki do gier, naturalnym wyborem dla tworzenia własnego silnika wydaje się być język C++.

3.1.2 Wybór biblioteki graficznej

Po wyborze języka programowania, kluczowym krokiem jest wybór technologii do tworzenia okna gry oraz rysowania obiektów graficznych. Na rynku dostępne są różne biblioteki graficzne, a kilka z nich wyróżnia się szczególnie: SFML, OpenGL, Allegro oraz Raylib.

SFML to biblioteka, która oferuje prostą i szybką integrację dla wielu elementów multimedialnych, co czyni ją popularnym wyborem dla początkujących. Posiada intuicyjne API i obsługuje zarówno grafikę, dźwięk, jak i obsługę wejścia.

Allegro to również znana biblioteka graficzna, zapewniająca szeroki zakres narzędzi do tworzenia gier. Jest stabilna i ma długi staż, co przyciąga programistów poszukujących solidnej i sprawdzonej biblioteki.

OpenGL to bardziej nisko poziomowe API graficzne, oferujące zaawansowane możliwości renderowania grafiki 2D i 3D. Jest często wykorzystywane w profesjonalnych projektach do stworzenia zaawansowanych efektów wizualnych.

Raylib jest kolejną interesującą opcją. Choć mniej popularny niż SFML czy Allegro, jest on prosty w użyciu i oferuje szybkie narzędzia do tworzenia gier i aplikacji. Ma minimalistyczne API i wsparcie dla wielu platform.

Z dostępnych możliwości OpengGL jest najniższą opcją co komplikuje tworzenie silnika dlatego ta opcja odpada z pozostałych 3 opcji najbardziej atrakcyjna jest biblioteka raylib mimo mniejszej popularności posiada dużo bardzo dobrych przykładów na stronie głównej z których zrozumienie pewnych elementów jest proste.

3.1.3 Wybór środowiska programowania

Na rynku istnieje kilka popularnych środowisk wspomagających programowanie w języku C++, oferujących czytelne interfejsy oraz możliwość dostosowania do indywidualnych potrzeb użytkowników.

CodeBlocks jest często wybierany przez początkujących programistów, ze względu na prostotę interfejsu i różnorodne narzędzia ułatwiające naukę programowania. Jego elastyczność i konfigurowalność pozwalają zarówno początkującym, jak i bardziej zaawansowanym programistom na wykorzystanie go w różnorodnych projektach.

Oprócz tego,

Visual Studio jest potężnym narzędziem oferującym szeroki zakres funkcji dla programistów C++. Dzięki zaawansowanym funkcjom edytora kodu, debuggerowi, narzędziom do profilowania oraz integracji z różnymi platformami, jest często wybierane przez profesjonalistów i stosowane w projektach komercyjnych.

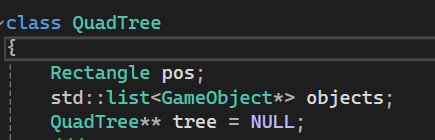
Natomiast CLion, skupia się na efektywnej pracy z językiem C++, oferując inteligentne wsparcie dla kodu, automatyczne uzupełnianie, analizę kodu oraz integrację z systemami kontroli wersji. Jest ceniony przez programistów za wydajność i przydatne narzędzia, które ułatwiają rozwój projektów. Jego interfejs jest bardziej wyspecjalizowany niż w przypadku CodeBlocks, co czyni go bardziej odpowiednim dla średnio zaawansowanych i zaawansowanych programistów.

Z podanych opcji najbardziej atrakcyjną opcją jest Visual Studio dzięki temu środowiskowi miedzy innymi za pomocą pakietów nuget można w prosty sposób zainstalować bibliotekę która nas interesuje do projektu.

3.2 Implementacja Drzewa Czwórkowego.

Wspominając podpunkt 2.3.1 obiekty które mogą zmieniać pozycję lub obiekty potrzebujące blok pod nim będą przechowywane w strukturze drzewa czwórkowego.

Dla tego problemu została zaimplementowana klasa QuadTree jako że klasa opiera się na strukturze drzewa. Klasa posiada podwójny wskaźnik na obiekt QuadTree o nazwie tree. Spowodowane jest to faktem że po otwarciu kolejnego poziomu drzewa tworzymy tablicę 4 elementów które dzielą obszar na kolejne 4 ćwiartki. Podwójny wskaźnik jest także spowodowany brakiem konstruktora domyślnego gdyż musimy podać pozycję w której drzewo funkcjonuje



Rysunek 3.2.1 Struktura drzewa w kodzie programu

Operacje na tej strukturze są rekurencyjne przez co nie mamy możliwości bezpośrednio odwołać się do danego kawałka drzew. Natomiast poprzez podanie odpowiednich fragmentów terenu z jakiego chcemy otrzymać obiekty struktura zwraca wszystkie obiekty odpowiadające danemu terytorium.



Rysunek 3.2.2 Metody drzewa czwórkowego

Drzewo posiada kilka dostępnych metod, które są używane przez inne obiekty w systemie. Pierwszą z nich jest konstruktor, który tworzy obiekt, natomiast drugą jest destruktor, który odpowiada za zwalnianie pamięci całej struktury.

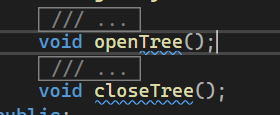
Przechodząc do często stosowanych metod po utworzeniu drzewa, istnieje metoda „addObj”, służąca do dodawania obiektu do odpowiedniej ćwiartki drzewa.

Z kolei metoda „removeObj” umożliwia usunięcie obiektu z struktury i jest często używana przy całkowitym wyeliminowaniu obiektu ze sceny lub w przypadku, gdy obiekt przemieści się zbyt daleko od konkretnego fragmentu mapy.

„UpdatePos” jest kolejną metodą, wykorzystywaną do aktualizacji pozycji obiektu w strukturze drzewa. Zazwyczaj stosuje się ją dla obiektów poruszających się po mapie. W momencie zmiany pozycji obiektu, ta metoda zostaje wywołana dla danego obiektu.

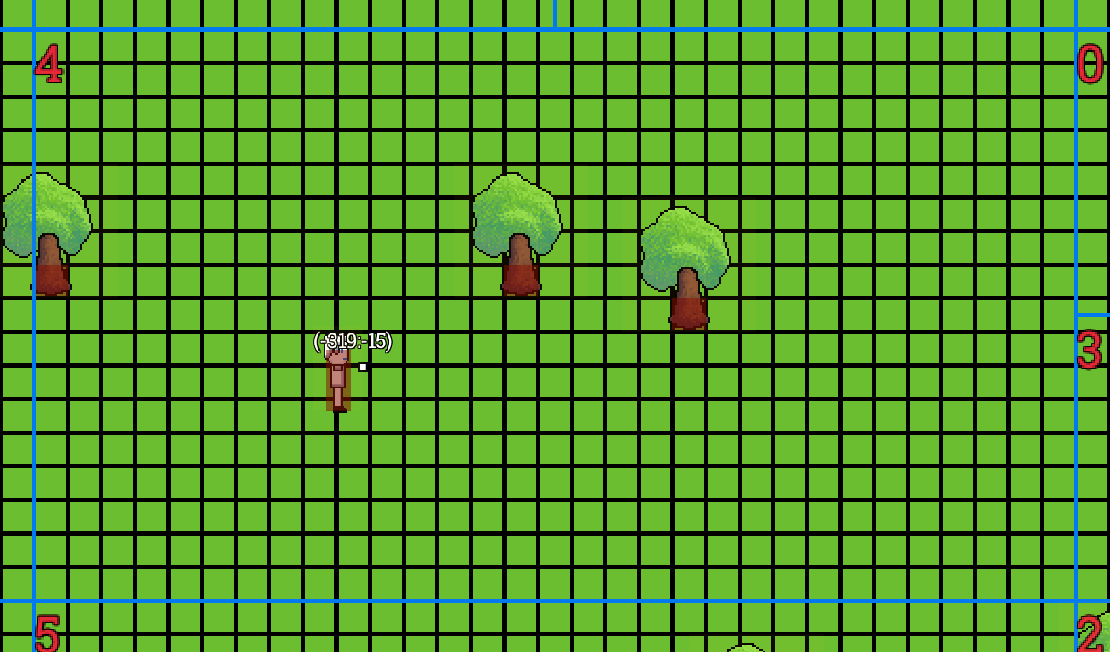
„HasObj” to kolejna metoda, która sprawdza, czy dany obiekt istnieje w drzewie. Natomiast metoda „draw” służy do rysowania struktury, co jest przydatne do sprawdzania poprawności funkcjonowania drzewa.

Dzięki tym różnorodnym metodom drzewo jest w stanie zarządzać obiektami i ich interakcjami w sposób elastyczny i efektywny.

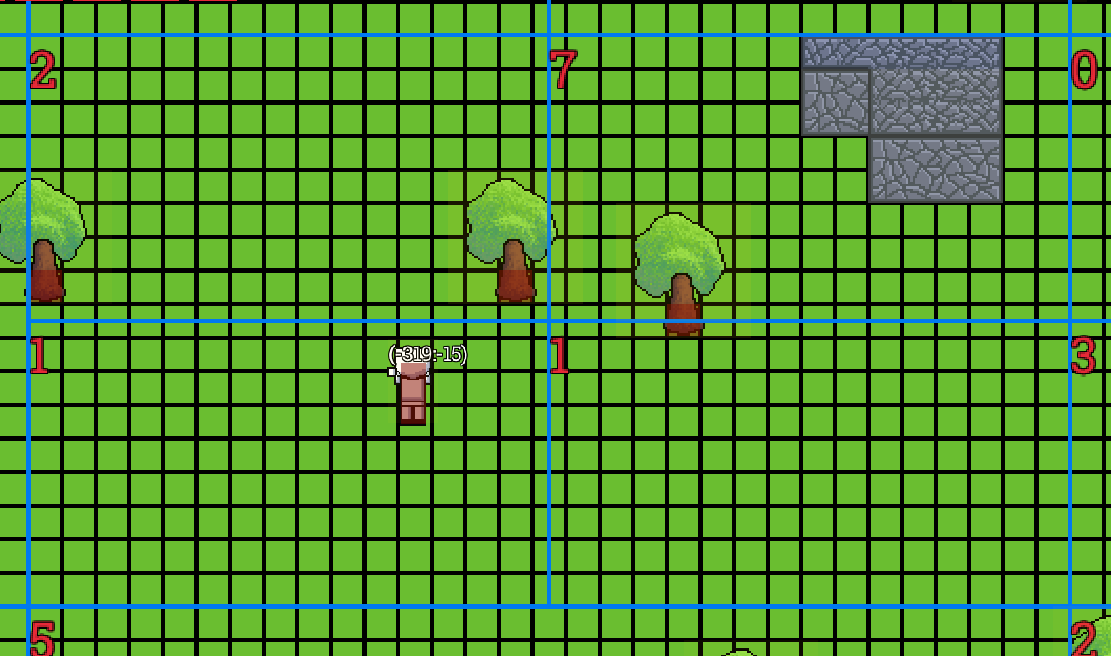


Rysunek 3.2.3 Prywatne metody klasy pozwalające zwiększyć drzewo oraz ucięcie fragmentu drzewa

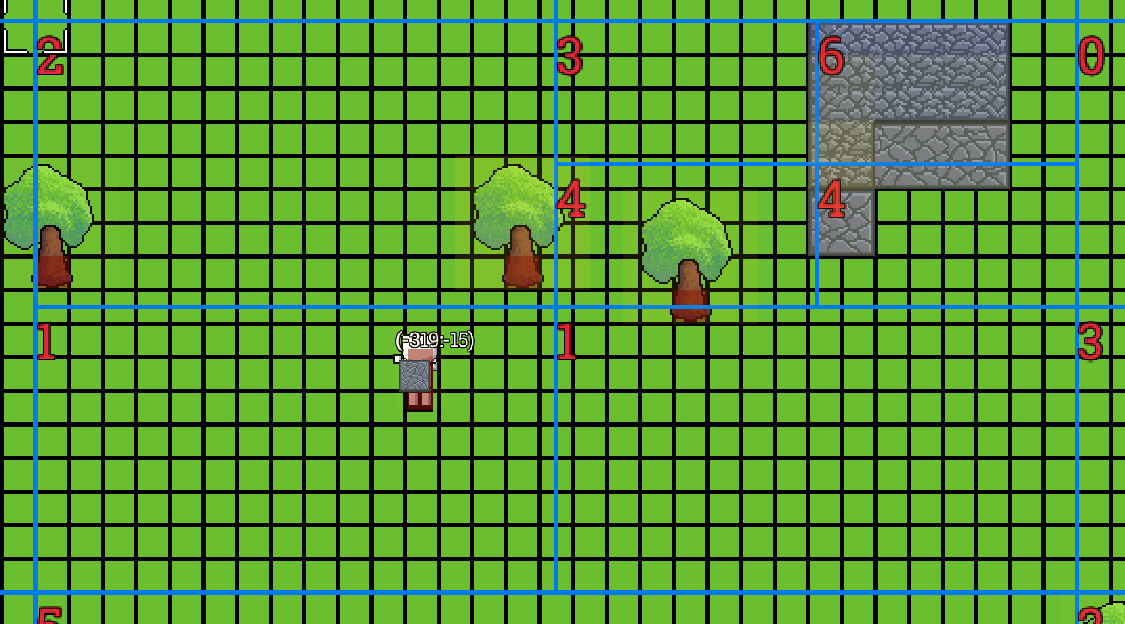
Istnieją jeszcze 2 metody prywatne które pomagają w rozszerzaniu struktury drzewa metoda „openTree” metoda ta dzieli obszar drzewa na 4 gdy w danym kawałku drzewa znajduje się więcej niż 8 obiektów. Natomiast metoda „closeTree” powoduje zamknięcie segmentu drzewa jeśli znajduje się tam mniej niż 4 obiekty by nie marnować pamięci na dużej ilości otwartych obszarów.



Rysunek 3.2.4 Przedstawienie struktury drzewa gdy w jego fragmencie znajduje się 8 obiektów.



Rysunek 3.2.5 Przedstawia strukturę drzewa po dodaniu dziewiątego obiektu.



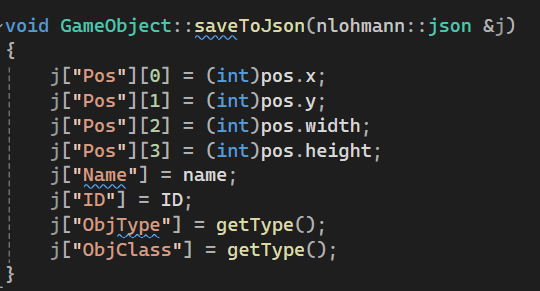
Rysunek 3.2.6 Przedstawienie struktury drzewa gdy zostanie dodane więcej obiektów w danej cwiardce.

Z obrazków można zaobserwować że z każdym kolejnym obiektem drzewo przechowujące obiekty zwiększa się. Czerwony napis oznacza ilość obiektów w ćwiartkach. Obiekty mogą wchodzić w kilka fragmentów drzewa gdyż ich wielkość jest wystarczająco duża by dane się powróżyły. Na rysunku 3.2.5 możemy zaobserwować że drzewo z prawej górnej ćwiartki znajduje się też w dolnej dlatego wartość dolnej ćwiartki wynosi 1.

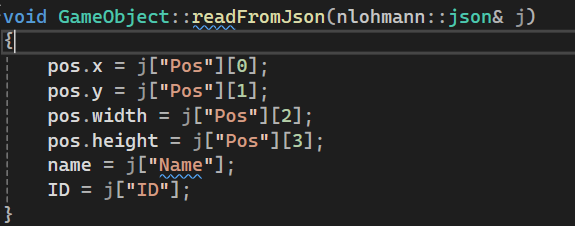
3.3 Implementacja Systemu zapisu

Zapisywanie danych z poszczególnych obiektów może stwarzać pewne wyzwania. Choć zapis obiektów w określonym, uporządkowanym formacie wydaje się rozsądny, dodawanie kolejnych zmiennych do obiektów może spowodować trudności w odczycie danych. Istnieją jednak sposoby na zapisywanie danych w sposób umożliwiający ich późniejsze odczytanie.

Obecnie powszechnie stosowanymi formatami plików do tego celu są JSON oraz XML. JSON jest elastycznym sposobem reprezentacji danych, dlatego też został wybrany do wykorzystania w projekcie. Posiada on także kluczową zaletę: jako że zapisuje dane w formie klucz-wartość, umożliwia nam sprawdzenie istnienia określonego klucza. Jest to istotne, ponieważ po wprowadzaniu zmian w kodzie lub dodawaniu nowych funkcji, pozwoli nam odczytać świat, który został wygenerowany w poprzednich wersjach aplikacji. Dzięki temu zachowamy spójność danych nawet w przypadku modyfikacji i ewolucji projektu.



Rysunek 3.3.1 Zapisywanie danych abstrakcyjnej klasy GameObject



Rysunek 3.3.2 Wczytywanie danych abstrakcyjnej klasy GameObject

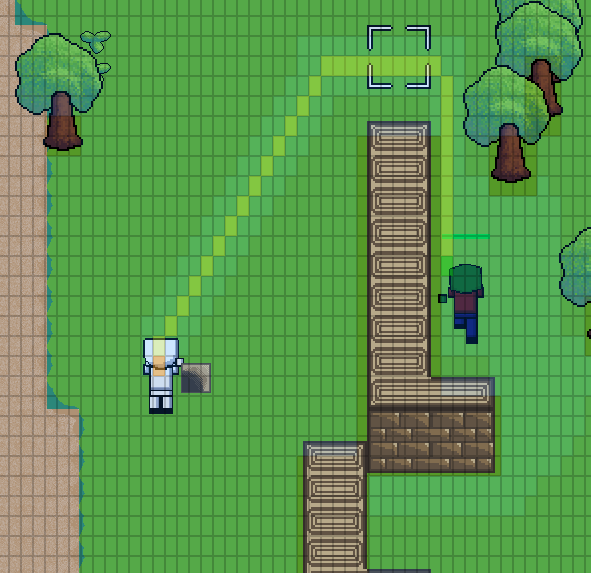
W naszej pierwotnej klasie z której wywodzą się wszystkie obiekty w grze istnieje wirtualna metoda do zapisywania i wczytywania danych do pliku Json. W kolejnych klasach dziedziczących dodaje się kolejne dane do zapisu i odczytu jednocześnie wywołujące funkcję pierwotną

3.4 Implementacja Systemu szukania drogi  
 Algorytmy wyszukiwania optymalnej ścieżki są powszechnie stosowane do tworzenia inteligencji poruszających się niezależnych postaci w grach. Istnieje wiele sposobów implementacji algorytmów wyszukiwania optymalnej ścieżki, jednak najczęściej używanym rozwiązaniem jest algorytm A\*.

Algorytm A\* opiera się na przemieszczaniu się przez kolejne połączenia punktów, zapisując czas potrzebny na dotarcie do danego punktu oraz odległość od celu, do którego dążymy. Jego złożoność nie jest zbyt skomplikowana, jednak pojawiają się problemy z jego wykorzystaniem w realnym scenariuszu.

Choć algorytm ten jest stosunkowo klarowny i wydajny, jego implementacja w praktycznych warunkach czasami napotyka na trudności związane z niestandardowymi scenariuszami i specyficznymi warunkami środowiska w grze.

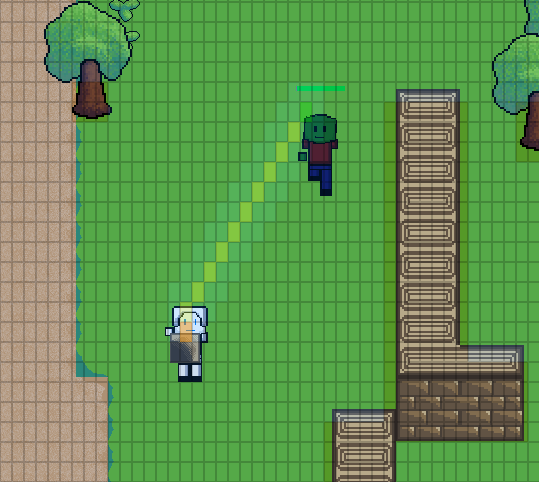
Początek formularza



Rysunek 3.4.1 Przeciwnik wyszukuje drogi do gracza.

Zgodnie z przedstawieniem na rysunku, przeciwnik ma wyznaczoną trasę, którą podąża, aby dotrzeć do gracza. Żółta ścieżka wyznacza bezpośrednią drogę do czerwonego prostokąta, symbolizującego pozycję gracza. Otaczająca jaśniejsza ścieżka ilustruje, które elementy drogi zostały wzięte pod uwagę podczas wyznaczania trasy przeciwnika.

W obszarze przy ścianie widoczne są bardziej wyblakłe oznaczenia, reprezentujące obszar, przez który przeciwnik nie jest w stanie przejść. To wyróżnienie wskazuje na nieprzejezdny teren dla przeciwnika, stanowiący przeszkodę na jego drodze do gracza.



3.4.2 Przeciwnik przeszedł przez wybudowaną przeszkodę

Po kilku sekundach obserwacji można zauważyć płynność ruchu przeciwnika bez żadnych trudności. To zoptymalizowane rozwiązanie, gotowe już w pierwszej wersji. W pierwotnej wersji, prostokąty reprezentujące przeciwnika były większe, co powodowało problemy z wyznaczaniem poprawnej ścieżki w przypadku przeszkód. Były to zaznaczone pola, które wprowadzały błędy w ścieżce poruszania się przeciwnika.

Jednakże, wyznaczanie ścieżki po każdym pojedynczym pikselu nie jest rozsądnym rozwiązaniem. Obliczenia i przechowywanie tak szczegółowej ścieżki w pamięci są zbyt kosztowne w stosunku do uzyskanego rezultatu. Dlatego podział postaci na elementy o wysokości i szerokości podzielonej na cztery części jest odpowiednim rozwiązaniem. Ten sposób działania działa właściwie i zapewnia optymalne rezultaty.



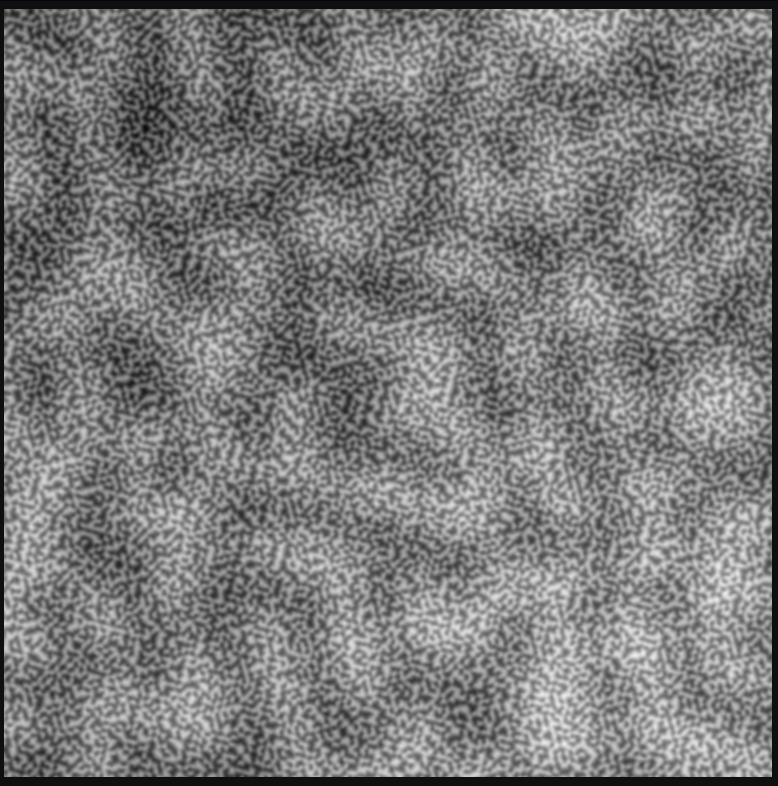
Rysunek 3.4.3 Pierwszy system szukania ścieżki

Z pierwotnej wersji wynika, że algorytm miał trudności z dokładnym określeniem obszarów, przez które nie można się poruszać. Zauważalne jest, że po lewej stronie od miejsca rozpoczęcia wyszukiwania trasy, istnieje możliwość przejścia w lewo. Jednakże, obszar ten jest zaznaczony jako wygaszony, co uniemożliwia dostęp do tego pola.

3.5 Generowanie terenu

Aby stworzyć świat generowany losowo z uporządkowaną strukturą, wykorzystuje się algorytmy generujące tzw. szum. Często używany jest szum Perlin, ale dla uzyskania interesującego terenu stosuje się również szum OpenSimplex2 ValueCubic. Biblioteka do generowania szumów jest dostępna na GitHubie[1].

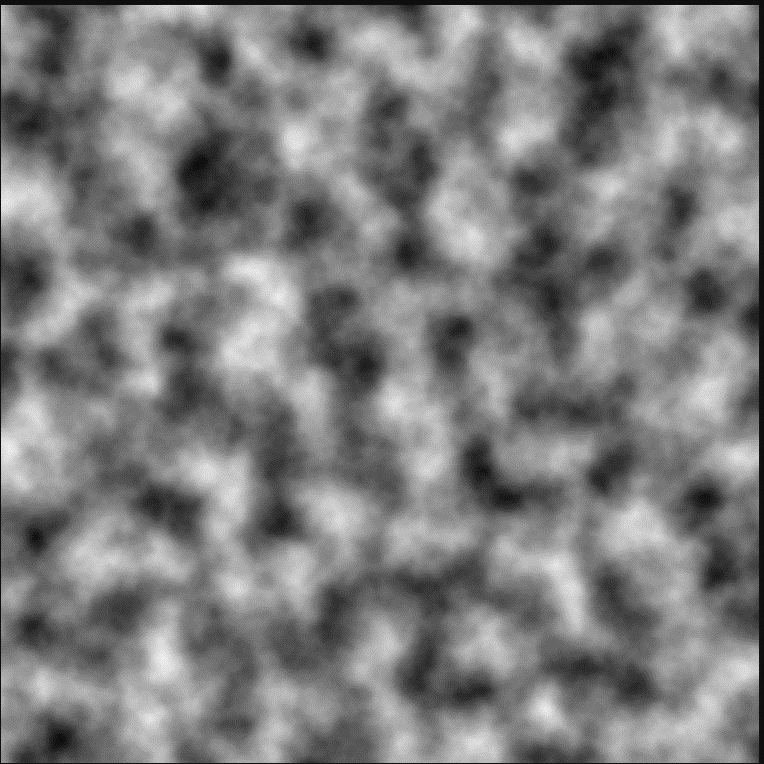
Korzystając z istniejącego rozwiązania do generowania szumów, konieczne jest właściwe dostosowanie parametrów szumów. Po ich modyfikacji uzyskano następujące rezultaty: Pierwszy szum został wygenerowany w celu określenia wartości terenu, który będzie determinował umiejscowienie obiektów na mapie, takich jak drzewa, kamienie czy inne rodzaje konstrukcji. Drugi szum określa obszary terenu oraz lokalizacje zbiorników wodnych.



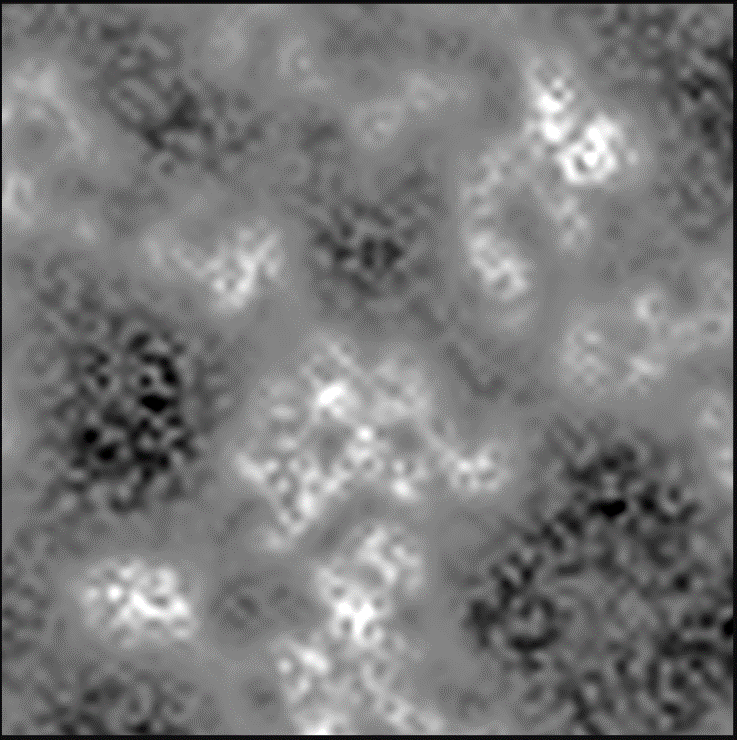
Rysunek 3.5.1 Szum odpowiedzialny za konstrukcje na mapie.



Rysunek 3.5.2 Szum odpowiedzialny za określenie występowania wody.

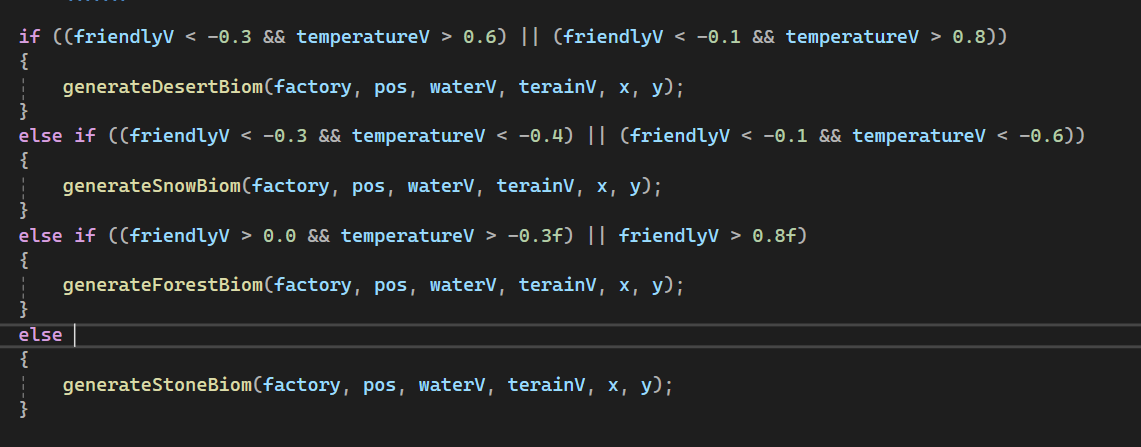


Rysunek 3.4.3 Szum odpowiedzialny za określenie przyjazności terenu.



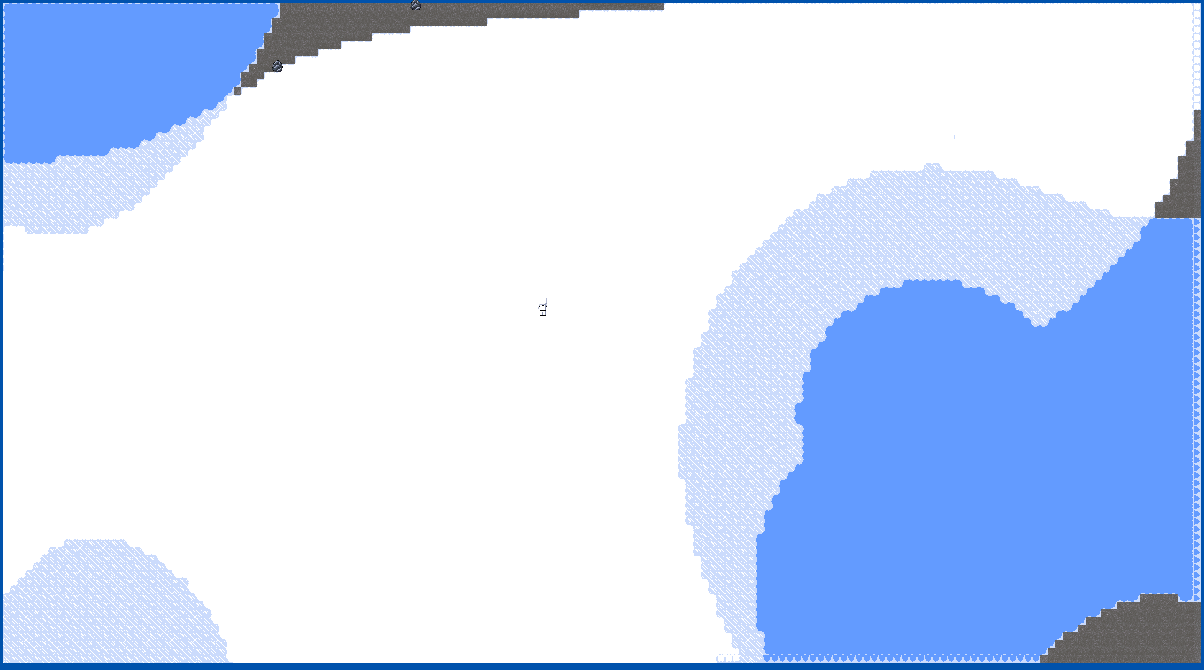
Rysunek 3.4.4 Szum odpowiedzialny za określenie temperatury występującej na danym obszarze

Trzeci szum odpowiada za uwarunkowania terenu, na którym przemieszczamy się. W przyszłości możliwe jest wykorzystanie tej wartości do regulacji ilości generowanych przeciwników, bądź też określenia obszarów, gdzie mogą pojawić się przyjazne istoty, takie jak zwierzęta lub niezależne postacie zajmujące się handlem.   
 Ostatni, czyli czwarty, szum zostanie wykorzystany do określenia temperatury obszarów. Dzięki temu będziemy w stanie zlokalizować na mapie obszary, gdzie mogą występować różne typy klimatyczne, takie jak pustynie, krainy ośnieżone czy zwykłe lasy.

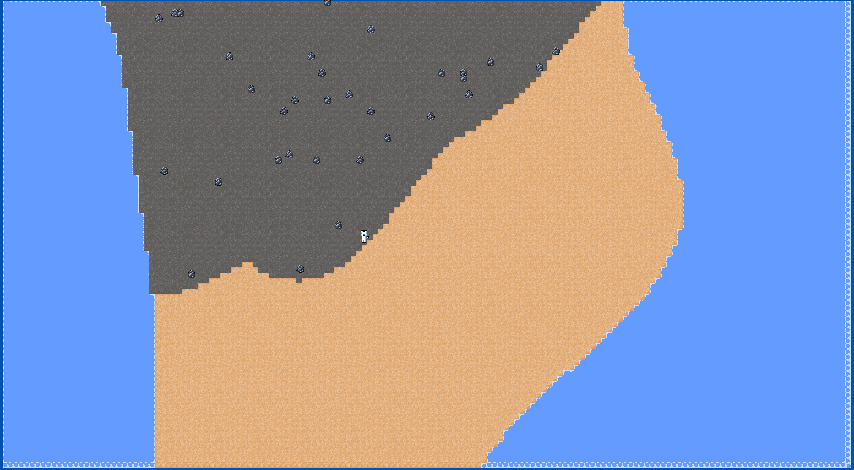


Rysunek 3.4.5 Fragment kodu determinujący teren jaki zostanie wygenerowany

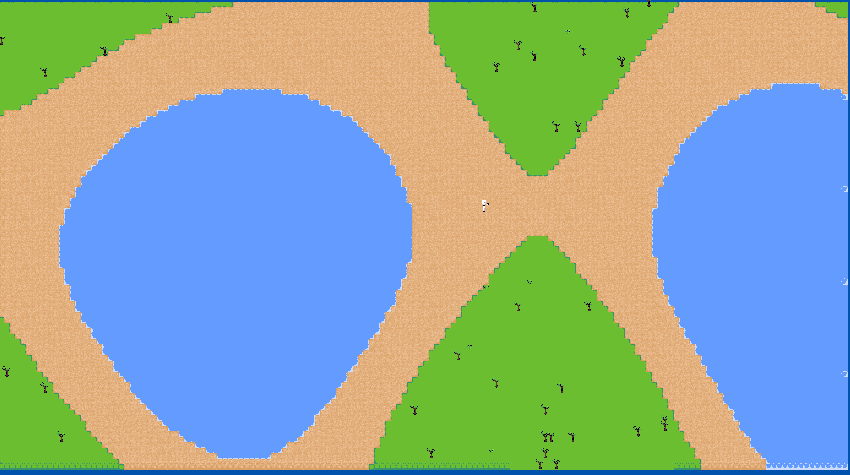
W kodzie, jak można zaobserwować na przedstawionym powyżej obrazku (Rysunek 3.4.5), wybór generowanego typu terenu zależy głównie od dwóch czynników: przyjazności terenu oraz temperatury. Przy niskiej przyjazności terenu mogą zostać wygenerowane biome piaskowe lub zimowe, zależnie jednak od temperatury, która odgrywa kluczową rolę w procesie generowania biomów. Im niższa temperatura, tym większe prawdopodobieństwo wygenerowania terenu zimowego, a im wyższa temperatura, tym większe prawdopodobieństwo wystąpienia pustyni. Należy również zaznaczyć, że wartości mieszczą się w przedziale od -1 do 1, gdzie -1 oznacza na przykład bardzo zimne miejsce, a 1 bardzo gorące. Jeśli teren jest przyjazny, zostanie wygenerowany biom leśny, natomiast jeśli żaden z powyższych warunków nie będzie spełniony, generowany będzie biom skalny. Efekty tego rozwiązania generuje teren taki jak przedstawiony poniżej.



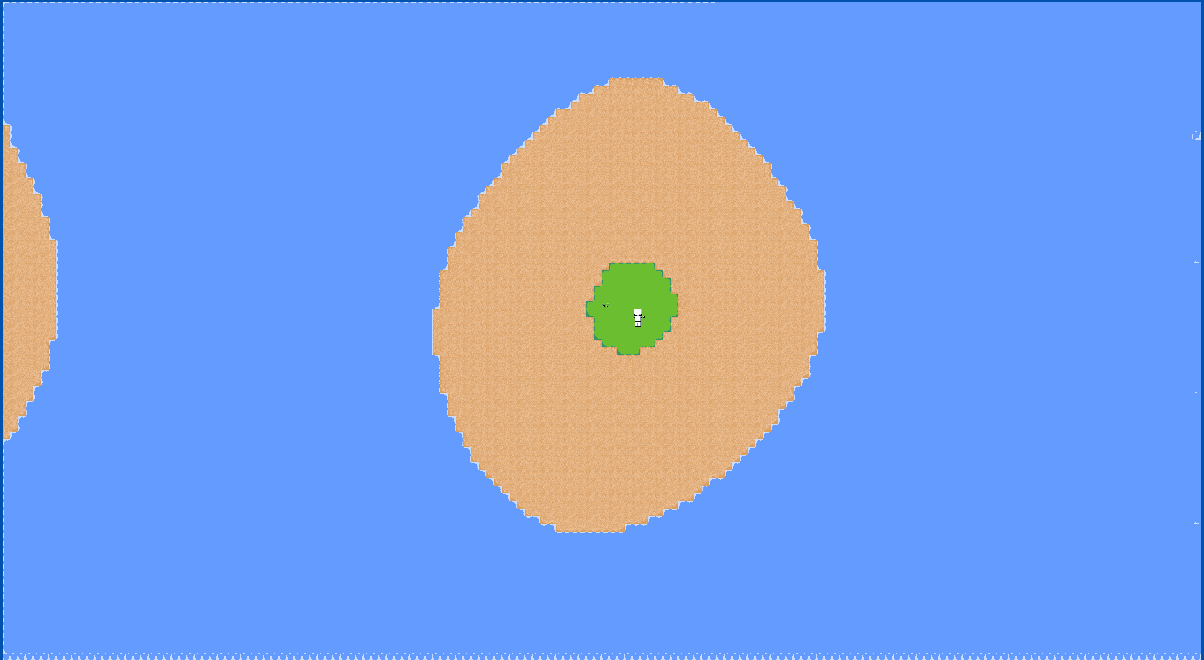
Rysunek 3.4.6 Wygenerowany teren ziomowy



Rysunek 3.4.7 Wygenerowana pustynia



Rysunek 3.4.8 Wygenerowany Las z plażą



Rysunek 3.4.9 Przykładowa wygenerowana wyspa

Bibliografia

1 Biblioteka szumów do generowania terenu.

https://github.com/Auburn/FastNoiseLite?tab=readme-ov-file